Algoritmia

Niveles esperados de Algoritmia

Niveles de desempeño	
Satisfactorio	Sobresaliente
El sustentante con un nivel de desempeño Satisfactorio es capaz de aplicar estructuras de datos lineales, analizar algoritmos con base en matemáticas discretas (bases numéricas, matrices, funciones, teoría de conjuntos, permutaciones, reglas de conteo y redes de Petri) y lógica computacional (lógica proposicional y lógica de primer orden, álgebra de Boole, circuitos lógicos); asimismo, puede valorar la integración de estos elementos para la solución de problemas en diversos ámbitos.	Además de lo señalado en el nivel Satisfactorio, el sustentante con nivel Sobresaliente es capaz de analizar problemas planteados con tablas *hash*, reglas de deducción, recursividad y combinaciones; también puede valorar las estructuras de datos no lineales, la representación de conocimiento usando lógica de predicados, la complejidad algorítmica y la inducción matemática en escenarios específicos.

Subtemas

1.1. Análisis y diseño de algoritmos

Definicion

- Un algoritmo es un conjunto ordenado y finito de operaciones que permite hallar la solución a un problema.
- **Mide la cantidad de recursos (tiempo y espacio)** que un algoritmo consume en función del tamaño de la entrada. Usamos notaciones como la Big O para expresar estos límites.
- **Técnicas comunes de diseño:Divide y vencerás, Programación Dinámica, Algoritmos Voraces.
- **Ejemplos clave:** Algoritmos de ordenación (burbuja, quicksort) y búsqueda (búsqueda binaria).

Notación Big O: Es una notación que describe el peor escenario posible en términos de tiempo o espacio. Por ejemplo, un algoritmo con una complejidad temporal de O(n) significa que, en el peor de los casos, el tiempo que tardará en ejecutarse es proporcional al tamaño de la entrada

Para analizar un algoritmo, se consideran los siguientes pasos:

1. **Definir el problema:** Comprender claramente el problema a resolver.
2. **Diseñar el algoritmo:** Crear un conjunto de pasos para resolver el problema.
3. **Implementar el algoritmo:** Codificarlo en un lenguaje de programación.
4. **Analizar el algoritmo:** Evaluar su complejidad temporal y espacial.

Ejercicio Práctico:

Considera el siguiente algoritmo simple: encontrar el elemento más grande en un arreglo de números.

```python
def max_element(arr):
    max_num = arr[0]
    for num in arr:
        if num > max_num:
            max_num = num
    return max_num
```

El algoritmo encuentra el numero mas grande de una lista de numeros, lo almacena y muestra, esto en base a un recorrido de un for y un condicional.

Complejidad Algorítmica

La **complejidad algorítmica** se refiere a cómo se comporta un algoritmo en términos de tiempo y/o espacio a medida que el tamaño de su entrada crece. Esta nos permite evaluar la eficiencia de un algoritmo.

Tipos de Complejidad

1. **Complejidad Temporal (Tiempo)**: Mide la cantidad de tiempo que un algoritmo tarda en ejecutarse en función del tamaño de la entrada.

2. **Complejidad Espacial (Espacio)**: Mide la cantidad de memoria que un algoritmo utiliza en función del tamaño de la entrada.

Notación Big O

- **O(1)**: Tiempo constante. Sin importar el tamaño de la entrada, el algoritmo siempre tardará el mismo tiempo.
- **O(log n)**: Tiempo logarítmico. Común en algoritmos que dividen a la entrada por la mitad en cada paso, como la búsqueda binaria.
- **O(n)**: Tiempo lineal. El tiempo que tarda el algoritmo crece linealmente con el tamaño de la entrada, como la búsqueda secuencial.
- **O(n log n)**: Tiempo log-lineal. Común en algoritmos eficientes de ordenación, como merge sort o quicksort.
- **O(n^2)**, **O(n^3)**, ...: Tiempo polinomial. Se presentan en algoritmos con bucles anidados.
- **O(2^n)**: Tiempo exponencial. Común en algoritmos que resuelven problemas al explorar muchos subproblemas, como el problema del viajante.

Análisis de Complejidad

Para analizar la complejidad de un algoritmo:

1. **Ignorar Operaciones Constantes**: Por ejemplo, si un algoritmo tiene 3 operaciones que toman tiempo constante y un bucle que toma tiempo n, su complejidad será O(n).
2. **Considerar el Peor Caso**: Analizar qué tan mal podría ser la situación.
3. **Identificar Bucles**: Por cada bucle, la complejidad subirá por un factor de n.

Notaciones de complejidad

Las notaciones más comunes para expresar la complejidad temporal son:

- **O**: Notación Big O. Representa el peor caso.
- **Ω**: Notación Omega. Representa el mejor caso.
- **Θ**: Notación Theta. Representa el caso promedio.

Ejemplos de ejercicios de complejidad

O(1): Tiempo constante

```python
def obtener_primer_elemento(lista):
    return lista[0]  # Acceso directo al primer elemento
```

Explicación: No importa el tamaño de `lista`, siempre toma el mismo tiempo acceder al primer elemento. La operación de acceso a un índice en una lista (o arreglo) es una operación de tiempo constante.

O(log n): Tiempo logarítmico

```python
def busqueda_binaria(arreglo, objetivo):
    izquierda, derecha = 0, len(arreglo) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if arreglo[medio] == objetivo:
            return medio
        elif arreglo[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1
```

Explicación: En cada iteración del bucle `while`, el tamaño del problema se reduce a la mitad (ya sea la parte izquierda o la derecha del arreglo), por lo que el número total de operaciones necesarias crece logarítmicamente con el tamaño del arreglo.

O(n): Tiempo lineal

```python
def busqueda_secuencial(lista, objetivo):
    for indice, elemento in enumerate(lista):
        if elemento == objetivo:
            return indice
    return -1
```

Explicación: Este bucle `for` recorre cada elemento de `lista` una vez, por lo que el tiempo de ejecución es directamente proporcional al número de elementos en `lista`.

O(n log n): Tiempo log-lineal

```python
def quicksort(arreglo):
    if len(arreglo) <= 1:
```

```
        return arreglo
    else:
        pivote = arreglo[len(arreglo) // 2]
        menores = [x for x in arreglo if x < pivote]
        mayores = [x for x in arreglo if x > pivote]
        return quicksort(menores) + [pivote] + quicksort(mayores)
```

Explicación: Quicksort divide el arreglo en cada llamada recursiva (comportamiento logarítmico) y luego realiza el proceso de combinación que tiene un costo lineal, resultando en una complejidad general de O(n log n).

O(n^2): Tiempo cuadrático

```
def ordenamiento_burbuja(arreglo):
    n = len(arreglo)
    for i in range(n):
        for j in range(0, n-i-1):
            if arreglo[j] > arreglo[j+1]:
                arreglo[j], arreglo[j+1] = arreglo[j+1], arreglo[j]
    return arreglo
```

Explicación: Hay dos bucles anidados que iteran sobre el arreglo, y en el peor de los casos, cada uno de ellos recorre `n` elementos, lo que resulta en O(n * n) = O(n^2).

O(2^n): Tiempo exponencial

```
def fibonacci_recursivo(n):
    if n <= 1:
        return n
    else:
        return fibonacci_recursivo(n-1) + fibonacci_recursivo(n-2)
```

Explicación: La función `fibonacci_recursivo` se llama a sí misma dos veces por cada llamada, creando un árbol de recursión que crece exponencialmente con `n`. Esto se debe a que para calcular `fibonacci_recursivo(n)`, primero se deben calcular `fibonacci_recursivo(n-1)` y `fibonacci_recursivo(n-2)`, y así sucesivamente.

Estos ejemplos ilustran cómo se determina la complejidad de tiempo de diferentes algoritmos basados en su estructura y la forma en que procesan los datos.

1.1.1 Pseudocódigo y Diagramas de Flujo

Pseudocódigo

El pseudocódigo es una forma de representar algoritmos utilizando un lenguaje intermedio entre el lenguaje natural y el lenguaje de programación. Su objetivo es representar de manera simple y comprensible para humanos el flujo y lógica del algoritmo.

Características del pseudocódigo:

- No está ligado a ningún lenguaje de programación en específico.
- Se utiliza lenguaje natural.
- Se escribe de manera estructurada y ordenada.

####Ejemplo

```
Inicio
   Leer numero1, numero2
   Suma = numero1 + numero2
   Imprimir Suma
Fin
```

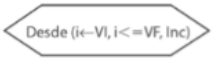
Diagramas de Flujo

Los diagramas de flujo son representaciones gráficas de algoritmos. Se utilizan símbolos predefinidos para representar diferentes acciones o pasos en el algoritmo. Es una herramienta muy útil para visualizar y entender el flujo lógico de un programa.

Símbolos comunes en los diagramas de flujo:

1. **Óvalo:** Indica el inicio o fin del algoritmo.
2. **Rectángulo:** Representa un proceso o acción que debe llevarse a cabo.
3. **Paralelogramo:** Indica una entrada o salida de datos.
4. **Rombo:** Representa una decisión y generalmente contiene una pregunta.
5. **Flechas:** Indican la dirección del flujo del proceso.

Representacion grafica

Símbolo	Descripción
	Inicio y final del diagrama de flujo.
	Entrada (leer) y salida de datos (imprimir).
	Símbolo de decisión. Indica la realización de una comparación de valores.
	Símbolo de proceso. Indica la asignación de un valor en la memoria y/o la ejecución de una operación aritmética.
	Líneas de flujo o dirección. Indican la secuencia en que se realizan las operaciones.
	Repetitiva desde número de iteraciones o repeticiones.
	Impresión
	Conectores

1.1.2 Programas Simples

1. Condicionales

Los condicionales se usan para tomar decisiones en el código.

```
Si (condición) Entonces
   // Código si la condición es verdadera
Sino
   // Código si la condición es falsa
Fin Si
```

2. Ciclos

Los ciclos se usan para repetir un bloque de código múltiples veces.

```
Para i = 1 Hasta 10 Hacer
   Imprimir i
Fin Para
```

3. Arreglos

Los arreglos son estructuras que permiten almacenar múltiples valores del mismo tipo. Pueden ser simples, paralelos o multidimensionales.

```
Definir numeros[5]
numeros[0] = 1
numeros[1] = 2
...
```

4. Funciones

Las funciones permiten segmentar el código en bloques reutilizables.

```
Función sumar(a, b)
   Retornar a + b
Fin Función

resultado = sumar(5, 3)  // resultado = 8
```

5. Parámetros

Los parámetros son los valores que se pasan a una función cuando se llama.

Ejemplo:

Usando la función anterior `sumar(a, b)`, `a` y `b` son parámetros.

1.1.3 Recursividad y Modularidad

1. Recursividad

La recursividad es una técnica en la cual una función se llama a sí misma. Se utiliza para resolver problemas que pueden ser divididos en problemas más pequeños del mismo tipo.

Ejemplo clásico: Factorial

El factorial de un número n (denotado como !n!) es el producto de todos los números positivos desde 1 hasta n. Y se puede definir recursivamente así:

- 0!0! = 1
- !n! = n×(n−1)!

Pseudocódigo de la función factorial recursiva:

```
Función factorial(n)
   Si n = 0 Entonces
      Retornar 1
   Sino
      Retornar n * factorial(n-1)
   Fin Si
Fin Función
```

---

2. Modularidad

La modularidad se refiere a la descomposición de un programa en módulos o partes más pequeñas (también llamadas funciones o procedimientos) que trabajan juntas. Al dividir el código en módulos, se puede reutilizar, es más legible y fácil de mantener.

Ejemplo: Calcular el área de diferentes figuras

Si tuviéramos que calcular el área de diferentes figuras, podríamos tener funciones separadas para cada figura.

Pseudocódigo de la modularidad para áreas:

```
Función areaCuadrado(lado)
   Retornar lado * lado
Fin Función

Función areaCirculo(radio)
   Retornar 3.14159 * radio * radio
Fin Función

Función areaRectangulo(base, altura)
   Retornar base * altura
Fin Función
```

Al diseñar algoritmos, la modularidad nos ayuda a segmentar y estructurar nuestro programa, permitiendo un enfoque más claro y enfocado en cada tarea específica. La recursividad, por otro lado, es útil cuando un problema puede ser dividido en subproblemas del mismo tipo.

1.2. Estructuras de datos

Definición:

- Una estructura de datos es una manera de organizar y almacenar datos para que puedan ser accedidos y modificados de manera eficiente.
- **Tipos básicos:** Arreglos, listas enlazadas, pilas, colas, árboles, grafos.
- **Operaciones:** Inserción, eliminación, búsqueda, ordenación, entre otros.

1.2.1 Estructuras de Datos Simples

Estructuras de Datos Simples:

Las estructuras de datos son formas de organizar y almacenar datos en una computadora para que puedan ser accedidos y modificados eficientemente. Veamos las estructuras simples:

1. **Variables**:
   - Son espacios en la memoria que almacenan valores que pueden cambiar durante la ejecución de un programa.
   - Tienen un tipo de dato asociado que define qué clase de información se puede almacenar en ellas: números enteros, flotantes, caracteres, booleanos, entre otros.
2. **Arreglos Simples**:
   - Es una colección de elementos, todos del mismo tipo, que están indexados por un entero.
   - Todos los elementos del arreglo están juntos en memoria.
   - Ejemplo: `int arr[5] = {1, 2, 3, 4, 5};`
3. **Matrices**:
   - Es un arreglo de dos dimensiones.
   - Se puede pensar en ello como una tabla con filas y columnas.
   - Ejemplo: `int matrix[2][2] = { {1, 2}, {3, 4} };`
4. **Vectores**:
   - Similar a un arreglo simple pero con la capacidad de cambiar su tamaño dinámicamente.
   - Los lenguajes de programación modernos, como C++ o Java, ofrecen bibliotecas para manejar vectores de manera eficiente.
   - Ejemplo en C++: `vector<int> vec = {1, 2, 3, 4, 5};`
5. **Arreglos Multidimensionales**:
   - Son arreglos que tienen más de dos dimensiones.
   - Ejemplo de un arreglo tridimensional: `int arr[2][2][2] = { {{1,2},{3,4}}, {{5,6},{7,8}} };`

El uso de estas estructuras depende del problema que estés tratando de resolver. Por ejemplo, una matriz podría ser útil si estás trabajando con sistemas de ecuaciones o transformaciones en gráficos por computadora. Los arreglos multidimensionales podrían ser útiles en aplicaciones como simulaciones o modelado 3D.

1.2.2 Pilas y colas

Pilas (Stacks):

Una pila es una estructura de datos en la que el último elemento que se agrega es el primero en ser removido (Last In, First Out - LIFO). Puedes pensar en una pila de platos; solo puedes agregar o quitar platos de la parte superior de la pila.

Operaciones básicas en una Pila:

- **Push:** Agrega un elemento a la parte superior de la pila.
- **Pop:** Elimina y devuelve el elemento superior de la pila.
- **Top/Peek:** Muestra el elemento superior sin eliminarlo.

Colas (Queues):

Una cola es una estructura de datos en la que el primer elemento que se agrega es el primero en ser removido (First In, First Out - FIFO). Imagina una fila de personas esperando en un banco; la primera persona en la fila es la primera en ser atendida.

Operaciones básicas en una Cola:

- **Enqueue:** Agrega un elemento al final de la cola.
- **Dequeue:** Elimina y devuelve el elemento del frente de la cola.
- **Front:** Muestra el elemento del frente sin eliminarlo.

Colas de Prioridad:

Una cola de prioridad es una extensión de una cola donde cada elemento tiene asignado una prioridad. Los elementos son atendidos según su prioridad y no el orden en el que fueron encolados.

Operaciones básicas en una Cola de Prioridad:

- **Insert:** Agrega un elemento con una prioridad dada.
- **DeleteMax (o DeleteMin):** Elimina el elemento con la prioridad más alta (o más baja).

¿Cuándo usar cada estructura?

- **Pila:** Cuando necesitas mantener un orden LIFO. Por ejemplo, en la implementación de algoritmos de backtracking o en la evaluación de expresiones

matemáticas (conversión de infijo a postfijo, evaluación de postfijo, etc.).

- **Cola:** Cuando necesitas mantener un orden FIFO. Por ejemplo, en algoritmos de búsqueda en anchura (BFS) o en la simulación de un servicio de atención al cliente.
- **Cola de Prioridad:** Cuando la prioridad es importante. Por ejemplo, en algoritmos de ruta más corta como Dijkstra o en la planificación de tareas en sistemas operativos.

Ejemplo pila:

```c
#include <stdio.h>
#include <stdbool.h>

#define MAX 10

typedef struct {
    int items[MAX];
    int top;
} Pila;

// Funciones de la Pila
void inicializar(Pila *p) {
    p->top = -1;
}

bool es_vacia(Pila *p) {
    return p->top == -1;
}

bool es_llena(Pila *p) {
    return p->top == MAX - 1;
}

void push(Pila *p, int item) {
    if(!es_llena(p)) {
        p->items[++(p->top)] = item;
    } else {
        printf("Pila llena.\n");
    }
}
```

```c
int pop(Pila *p) {
    if(!es_vacia(p)) {
        return p->items[(p->top)--];
    } else {
        printf("Pila vacía.\n");
        return -1;
    }
}

int peek(Pila *p) {
    if(!es_vacia(p)) {
        return p->items[p->top];
    } else {
        printf("Pila vacía.\n");
        return -1;
    }
}

int main() {
    Pila pila;
    inicializar(&pila);
    push(&pila, 1);
    push(&pila, 2);
    push(&pila, 3);
    printf("%d\n", peek(&pila));
    printf("%d\n", pop(&pila));
    printf("%d\n", pop(&pila));
    printf("%d\n", pop(&pila));
    return 0;
}
```

Ejemplo cola:

```c
#include <stdio.h>
#include <stdbool.h>

#define MAX 10

typedef struct {
    int items[MAX];
    int front, rear;
} Cola;

// Funciones de la Cola
void inicializar(Cola *c) {
    c->front = 0;
    c->rear = -1;
}

bool es_vacia(Cola *c) {
    return c->rear < c->front;
}

bool es_llena(Cola *c) {
    return c->rear == MAX - 1;
}
```

```c
void enqueue(Cola *c, int item) {
    if(!es_llena(c)) {
        c->items[++(c->rear)] = item;
    } else {
        printf("Cola llena.\n");
    }
}

int dequeue(Cola *c) {
    if(!es_vacia(c)) {
        return c->items[(c->front)++];
    } else {
        printf("Cola vacía.\n");
        return -1;
    }
}

int main() {
    Cola cola;
    inicializar(&cola);
    enqueue(&cola, 1);
    enqueue(&cola, 2);
    enqueue(&cola, 3);
    printf("%d\n", dequeue(&cola));
    printf("%d\n", dequeue(&cola));
    printf("%d\n", dequeue(&cola));
    return 0;
}
```

1.2.3 Listas Enlazadas y sus Tipos

Tipos de Listas Enlazadas:

1. **Lista Enlazada Simple**: Cada nodo tiene un único enlace que apunta al siguiente nodo en la secuencia. El último nodo apunta a `NULL`, indicando el final de la lista.
2. **Lista Enlazada Doble**: Cada nodo tiene dos enlaces, uno apunta al siguiente nodo y otro al anterior. Esto facilita la iteración en ambas direcciones.

3. **Lista Enlazada Circular**: Similar a la lista enlazada simple, pero el último nodo en lugar de apuntar a `NULL`, apunta de nuevo al primer nodo, formando un círculo.
4. **Lista Enlazada Doble Circular**: Combina las características de las listas dobles y circulares. El último nodo apunta al primero y el primero al último, permitiendo una iteración circular en ambas direcciones.

Cuándo usar cada tipo:

- **Lista Enlazada Simple**: Cuando solo necesitas iterar en una dirección y quieres una implementación simple con un uso de memoria eficiente.
- **Lista Enlazada Doble**: Cuando necesitas iterar en ambas direcciones o eliminar nodos de manera eficiente sin tener que recorrer toda la lista para encontrar el nodo anterior.
- **Lista Enlazada Circular**: Útil en aplicaciones donde necesitas un acceso cíclico a los datos, como en la gestión de recursos compartidos o en algoritmos de planificación.
- **Lista Enlazada Doble Circular**: Útil cuando se requieren las ventajas de una lista doble y además se necesita iterar los elementos de manera cíclica.

Lista Enlazada Simple

```
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None

# Ejemplo de creación y adición de nodos
nodo1 = Nodo(1)
nodo2 = Nodo(2)
nodo1.siguiente = nodo2  # El nodo1 apunta al nodo2
```

Explicación: Cada nodo tiene un campo de datos y un enlace a `siguiente`. La lista termina cuando un nodo apunta a `None`.

Lista Enlazada Doble

```
class NodoDoble:
    def __init__(self, dato):
```

```
        self.dato = dato
        self.siguiente = None
        self.anterior = None

# Ejemplo de creación y adición de nodos
nodo1 = NodoDoble(1)
nodo2 = NodoDoble(2)
nodo1.siguiente = nodo2  # El nodo1 apunta hacia adelante al nodo2
nodo2.anterior = nodo1  # El nodo2 apunta hacia atrás al nodo1
```

Explicación: Cada nodo tiene un campo de datos, un enlace a `siguiente` y otro a `anterior`, permitiendo recorridos en ambas direcciones.

Lista Enlazada Circular

```
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None

# Ejemplo de creación y adición de nodos
nodo1 = Nodo(1)
nodo2 = Nodo(2)
nodo1.siguiente = nodo2
nodo2.siguiente = nodo1  # Crea un círculo apuntando de nuevo al nodo1
```

Explicación: Funciona como una lista enlazada simple pero el último nodo apunta de nuevo al primero, creando un bucle.

Lista Enlazada Doble Circular

```
class NodoDobleCircular:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None
        self.anterior = None

# Ejemplo de creación y adición de nodos
nodo1 = NodoDobleCircular(1)
nodo2 = NodoDobleCircular(2)
nodo1.siguiente = nodo2
```

```
nodo2.siguiente = nodo1  # Crea una conexión circular hacia adelante
nodo1.anterior = nodo2
nodo2.anterior = nodo1  # Crea una conexión circular hacia atrás
```

Explicación: Similar a la lista enlazada doble pero en este caso el último nodo apunta al primero con su enlace `siguiente` y el primero al último con su enlace `anterior`, lo que permite iterar la lista en cualquier dirección de manera indefinida.

1.2.4 Arboles y sus tipos

Los árboles son estructuras de datos no lineales que simulan una jerarquía con relaciones padre-hijo. Cada nodo del árbol puede tener cero o más nodos hijos, y un solo nodo padre, excepto la raíz que no tiene padre.

Tipos de Árboles:

1. **Árbol Binario**: Cada nodo tiene como máximo dos hijos, conocidos como hijo izquierdo y derecho.
2. **Árbol Binario de Búsqueda (BST)**: Es un árbol binario con la propiedad de que para cada nodo, todos los elementos en su subárbol izquierdo son menores que el nodo, y todos los elementos en su subárbol derecho son mayores.
3. **Árbol AVL**: Es un árbol binario de búsqueda balanceado donde la diferencia de alturas entre el subárbol izquierdo y derecho para cualquier nodo es como máximo uno.
4. **Árbol Rojo-Negro**: Es otro tipo de árbol binario de búsqueda balanceado que utiliza colores (rojo y negro) en los nodos para asegurar que el árbol se mantenga equilibrado durante las inserciones y eliminaciones.
5. **Árbol B**: Es un árbol balanceado de búsqueda que puede tener más de dos hijos. Está optimizado para sistemas que leen y escriben grandes bloques de datos. Es comúnmente usado en bases de datos y sistemas de archivos.
6. **Árbol B+**: Es una variante del árbol B donde todos los valores están en las hojas y los nodos internos solo sirven como guías para llegar a estos valores. Los nodos hoja están enlazados, lo que facilita las operaciones de rango y secuenciales.
7. **Árbol Binario Completo**: Es un árbol binario en el que todos los niveles, excepto posiblemente el último, están completamente llenos, y todos los nodos están lo más a la izquierda posible.

8. **Heap (Montículo)**: Es un árbol binario especial donde el nodo padre es mayor (en un max-heap) o menor (en un min-heap) que los nodos hijos. Se utiliza para implementar colas de prioridad.

Algoritmos Propios de los Árboles:

- **Recorrido**: Inorden, Preorden, Postorden, y por niveles (BFS).
- **Búsqueda Binaria**: En un BST, permite encontrar elementos en tiempo logarítmico.
- **Inserción y Eliminación**: En BST, AVL, Rojo-Negro, y árboles B/B+.
- **Balanceo**: En AVL y Rojo-Negro para mantener el árbol equilibrado después de inserciones y eliminaciones.

Cuándo usar cada tipo de Árbol:

- **BST**: Cuando se requiere una búsqueda rápida y el conjunto de datos no cambia frecuentemente.
- **AVL**: Cuando se requiere que el árbol esté balanceado y las búsquedas son más frecuentes que las inserciones/eliminaciones.
- **Rojo-Negro**: Cuando se requiere balanceo y las inserciones/eliminaciones son tan frecuentes como las búsquedas.
- **Árbol B/B+**: En aplicaciones de bases de datos y sistemas de archivos donde se manejan grandes cantidades de datos y se necesita minimizar el acceso a disco.
- **Heap**: Cuando se requieren operaciones de extracción del máximo o mínimo elemento de manera eficiente, como en algoritmos de ordenamiento por montículo (heapsort) o colas de prioridad.

A continuación se presentan ejemplos en código de Python para cada tipo de árbol mencionado, con una explicación de sus características y propiedades particulares:

Árbol Binario

```
class NodoBinario:
    def __init__(self, valor):
        self.valor = valor
        self.izquierdo = None
        self.derecho = None

# Creación de un árbol binario simple
```

```
raiz = NodoBinario(2)
raiz.izquierdo = NodoBinario(1)
raiz.derecho = NodoBinario(3)
```

Explicación: En un árbol binario, cada nodo tiene hasta dos hijos. En este ejemplo, el nodo con valor `2` es la raíz, con un hijo izquierdo `1` y un hijo derecho `3`.

Árbol Binario de Búsqueda (BST)

```
class NodoBST:
    def __init__(self, valor):
        self.valor = valor
        self.izquierdo = None
        self.derecho = None

    def insertar(self, valor):
        if valor < self.valor:
            if self.izquierdo is None:
                self.izquierdo = NodoBST(valor)
            else:
                self.izquierdo.insertar(valor)
        else:
            if self.derecho is None:
                self.derecho = NodoBST(valor)
            else:
                self.derecho.insertar(valor)

# Creación de un BST
raiz = NodoBST(10)
raiz.insertar(5)
raiz.insertar(15)
```

Explicación: En un BST, todos los nodos del subárbol izquierdo de cualquier nodo tienen valores menores, y todos los nodos del subárbol derecho tienen valores mayores. Aquí, el `5` se inserta a la izquierda de `10`, y `15` a la derecha.

Árbol AVL

```
class NodoAVL(NodoBST):
    def __init__(self, valor):
        super().__init__(valor)
```

```
        self.altura = 1  # Altura inicial para el nodo

    # La inserción en un AVL requiere lógica adicional para mantenerlo
balanceado.
    # Aquí se simplifica y no se incluye el código completo para balancear
el árbol.
    # Se requieren rotaciones para mantener el balance después de
inserciones y eliminaciones.
```

Explicación: Un AVL extiende un BST y mantiene el balance para asegurarse de que la diferencia de altura entre subárboles izquierdo y derecho de cualquier nodo no sea más de uno.

Árbol Rojo-Negro

```
class NodoRojoNegro(NodoBST):
    def __init__(self, valor, color="rojo"):
        super().__init__(valor)
        self.color = color
        self.padre = None

    # Como en el caso del AVL, la inserción y eliminación en un árbol rojo-
negro requiere
    # mantener el balance mediante rotaciones y cambios de color.
```

Explicación: Un árbol rojo-negro utiliza colores y reglas adicionales en los nodos para asegurar que el árbol permanezca equilibrado después de inserciones y eliminaciones. Esto permite que las operaciones de búsqueda, inserción y eliminación se realicen en tiempo logarítmico.

Árbol B

```
class NodoB:
    def __init__(self, t):
        self.claves = []
        self.hijos = []
        self.t = t  # El grado mínimo del árbol B

    # Los métodos para inserción en un árbol B son más complejos que los de
```

```
un BST y requieren
    # dividir nodos cuando alcanzan el número máximo de claves.
```

Explicación: Un árbol B es un árbol de búsqueda auto-balanceado que puede tener un alto número de hijos por nodo. Está optimizado para sistemas que leen y escriben grandes bloques de datos, como las bases de datos.

Árbol B+

```
class NodoBPlus(NodoB):
    def __init__(self, t):
        super().__init__(t)
        self.siguiente = None  # Enlace al siguiente nodo hoja

    # Los nodos hoja de un árbol B+ contienen los datos y están enlazados
entre sí, lo que facilita
    # el recorrido secuencial de los valores.
```

Explicación: En un árbol B+, todas las claves

se almacenan en nodos hoja que están enlazados entre sí, lo que permite un recorrido eficiente de los datos en un rango específico.

Árbol Binario Completo

```
# Utilizando la misma clase NodoBinario definida anteriormente.

# Para crear un árbol binario completo, simplemente seguimos añadiendo
nodos asegurándonos
# de que cada nivel está completamente lleno antes de pasar al siguiente.
```

Explicación: Un árbol binario completo es aquel en el que cada nivel, excepto quizá el último, está completamente lleno, y todos los nodos están tan a la izquierda como sea posible.

Heap (Montículo)

```
import heapq

# Python proporciona una implementación de heap con su módulo heapq.
```

```
heap = []
heapq.heappush(heap, (10, "diez"))  # Inserta en un min-heap
heapq.heappush(heap, (1, "uno"))
valor = heapq.heappop(heap)  # valor es (1, "uno"), el elemento más pequeño
```

Explicación: Un montículo es un árbol binario completo donde los nodos siguen la propiedad del montículo: en un max-heap, cada nodo padre es mayor que sus nodos hijos; en un min-heap, cada nodo padre es menor que sus hijos. Esto permite acceso rápido al elemento máximo o mínimo.

1.2.5 Grafos

Características y Componentes de los Grafos:

Un **grafo** G se define como un conjunto de **nodos** (o vértices) y un conjunto de **aristas** (o bordes) que conectan pares de nodos. Las características y componentes principales de un grafo incluyen:

- **Nodos (Vértices)**: Son los puntos que componen el grafo.
- **Aristas (Bordes)**: Son las conexiones entre los nodos.
- **Adyacencia**: Dos nodos están adyacentes si están conectados por una arista.
- **Grado**: El número de aristas conectadas a un nodo.
- **Camino**: Una secuencia de nodos en la que cada nodo adyacente está conectado por una arista.
- **Ciclo**: Un camino que comienza y termina en el mismo nodo.
- **Conexo**: Un grafo es conexo si hay un camino entre cualquier par de nodos.
- **Subgrafo**: Un grafo formado a partir de un subconjunto de nodos y aristas de un grafo mayor.
- **Grafo dirigido (Digrafo)**: Las aristas tienen una dirección definida.
- **Grafo no dirigido**: Las aristas no tienen dirección.

Tipos de Grafos:

- **Dirigidos**: Las aristas tienen una dirección.
- **No dirigidos**: Las aristas no tienen dirección.
- **Pesados**: Las aristas tienen pesos o costos asociados.
- **Simple**: No hay bucles (aristas conectadas a sí mismas) ni múltiples aristas entre el mismo conjunto de nodos.

- **Completo**: Cada par de nodos está conectado por una arista.
- **Bipartito**: Se puede dividir los nodos en dos conjuntos, donde las aristas sólo conectan nodos de conjuntos diferentes.
- **Árbol**: Un grafo conexo sin ciclos.
- **Acíclico Dirigido (DAG)**: Un grafo dirigido sin ciclos.

Formas de Representación de Grafos:

- **Matriz de Adyacencia**: Una matriz cuadrada M donde M[i][j] representa la presencia (y a menudo el peso) de una arista del nodo i al nodo j.
- **Lista de Adyacencia**: Una lista que asocia a cada nodo con una lista de otros nodos a los que está conectado.
- **Matriz de Incidencia**: Una matriz que relaciona nodos y aristas, donde las filas representan nodos y las columnas representan aristas, indicando qué nodo es incidido por qué arista.

Algoritmos en Grafos:

- **Algoritmo de Dijkstra**: Encuentra el camino más corto desde un nodo a todos los demás en un grafo con pesos no negativos.
- **Algoritmo de Floyd-Warshall**: Encuentra la distancia más corta entre todos los pares de nodos en un grafo con pesos positivos o negativos (pero sin ciclos negativos).
- **Algoritmo de Prim**: Encuentra un árbol de expansión mínima para un grafo conexo con pesos.
- **Algoritmo de Kruskal**: También construye un árbol de expansión mínima, pero usando un enfoque basado en aristas.
- **Algoritmo de Bellman-Ford**: Calcula el camino más corto en grafos con pesos negativos y puede detectar ciclos negativos.
- **Búsqueda en Anchura (BFS)**: Para recorrer o buscar en un grafo nivel por nivel.
- **Búsqueda en Profundidad (DFS)**: Para recorrer o buscar en un grafo siguiendo un camino hasta el final antes de retroceder.

Aplicaciones y Selección de Tipo de Grafo:

La elección de un tipo de grafo depende del problema específico a resolver:

- **Redes Sociales**: Se suelen modelar con grafos no dirigidos para representar la conexión entre personas.
- **Sistemas de Rutas**: Utilizan grafos dirigidos y pesados para representar rutas con direcciones y distancias.
- **Árboles de Decisión**: Se modelan con grafos acíclicos dirigidos.
- **Organigrama de una Empresa**: A menudo se representa como un árbol o DAG.

Claro, veamos cada uno de estos algoritmos con un ejemplo y una explicación:

1. Algoritmo de Dijkstra

Explicación:
El algoritmo de Dijkstra es utilizado para encontrar la ruta más corta desde un nodo específico a todos los demás nodos en un grafo con pesos no negativos en las aristas. Funciona de la siguiente manera:

- Se asigna a cada nodo una distancia temporal: 0 para el nodo inicial y ∞ (infinito) para todos los demás nodos.
- Se selecciona el nodo con la menor distancia temporal, se actualizan las distancias de sus vecinos y se marca como "visitado".
- Se repite el proceso para todos los nodos no visitados del grafo.

Ejemplo:
Imagina un mapa de una ciudad con intersecciones como nodos y las calles como aristas. Las distancias entre intersecciones son los pesos de las aristas. Empezando desde una intersección específica, Dijkstra calcula la distancia más corta a todas las otras intersecciones.

1. Algoritmo de Dijkstra en Python

```
import sys

def dijkstra(graph, start):
    shortest_distance = {}
    predecessor = {}
    unseenNodes = graph
    infinity = sys.maxsize
    path = []
    for node in unseenNodes:
        shortest_distance[node] = infinity
```

```
    shortest_distance[start] = 0

    while unseenNodes:
        minNode = None
        for node in unseenNodes:
            if minNode is None:
                minNode = node
            elif shortest_distance[node] < shortest_distance[minNode]:
                minNode = node

        for childNode, weight in graph[minNode].items():
            if weight + shortest_distance[minNode] <
shortest_distance[childNode]:
                shortest_distance[childNode] = weight +
shortest_distance[minNode]
                predecessor[childNode] = minNode
        unseenNodes.pop(minNode)

    currentNode = end
    while currentNode != start:
        try:
            path.insert(0, currentNode)
            currentNode = predecessor[currentNode]
        except KeyError:
            print('Path not reachable')
            break
    path.insert(0, start)
    if shortest_distance[end] != infinity:
        print('Shortest distance is ' + str(shortest_distance[end]))
        print('And the path is ' + str(path))

# Ejemplo de uso
graph = {'A': {'B': 1, 'C': 4},
         'B': {'A': 1, 'C': 2, 'D': 5},
         'C': {'A': 4, 'B': 2, 'D': 1},
         'D': {'B': 5, 'C': 1}}
start = 'A'
end = 'D'
dijkstra(graph, start)
```

2. Algoritmo de Floyd-Warshall

Explicación:
El algoritmo de Floyd-Warshall es utilizado para encontrar las distancias más cortas entre todos los pares de nodos en un grafo. A diferencia de Dijkstra, puede manejar pesos negativos en las aristas, pero no ciclos negativos. Funciona de la siguiente manera:

- Inicialmente, la matriz de distancias se llena con los pesos directos de las aristas entre cada par de nodos.
- Luego, se itera sobre todos los nodos del grafo y se actualiza la matriz de distancias considerando si pasar por un nodo intermedio reduce la distancia entre cada par de nodos.

Ejemplo:
En una red de transporte público, donde las estaciones son nodos y las rutas son aristas con tiempos de viaje como pesos (incluyendo posibles tiempos negativos, como descuentos de tiempo por rutas rápidas), Floyd-Warshall encuentra el tiempo más corto posible entre todas las estaciones.

2. Algoritmo de Floyd-Warshall en Python

```
def floyd_warshall(graph):
    n = len(graph)
    distance = list(map(lambda i: list(map(lambda j: j, i)), graph))

    for k in range(n):
        for i in range(n):
            for j in range(n):
                distance[i][j] = min(distance[i][j], distance[i][k] +
distance[k][j])

    return distance

# Ejemplo de uso
graph = [[0, 5, float('inf'), 10],
         [float('inf'), 0, 3, float('inf')],
         [float('inf'), float('inf'), 0, 1],
         [float('inf'), float('inf'), float('inf'), 0]]
print(floyd_warshall(graph))
```

3. Algoritmo de Prim

Explicación:
El algoritmo de Prim se utiliza para encontrar un árbol de expansión mínima en un grafo conexo con pesos. Esto significa que encuentra un subconjunto de aristas que conecta todos los nodos del grafo con el menor costo total posible y sin ciclos. Funciona de la siguiente manera:

- Se comienza con un nodo arbitrario y se agrega al árbol de expansión.
- En cada paso, se agrega la arista más ligera que conecta un nodo en el árbol con un nodo fuera de él.
- Se repite este proceso hasta que todos los nodos estén en el árbol.

Ejemplo:
Considera un proyecto de tendido de cableado de fibra óptica donde cada nodo es una central y las aristas representan el posible cableado entre ellas. El algoritmo de Prim determinaría el camino más eficiente de cableado que conecta todas las centrales con el menor costo total.

3. Algoritmo de Prim en Python

```
import sys

def prim(graph, start):
    selected = [start]
    num_nodes = len(graph)
    no_edge = 0
    mincost = 0

    while no_edge < num_nodes - 1:
        minimum = sys.maxsize
        a = b = 0
        for m in selected:
            for n in range(num_nodes):
                if graph[m][n] and n not in selected:
                    if minimum > graph[m][n]:
                        minimum = graph[m][n]
                        a, b = m, n
        selected.append(b)
        no_edge += 1
        mincost += graph[a][b]

    return mincost
```

```
# Ejemplo de uso
graph = [[0, 2, 3, 0, 0, 0],
         [2, 0, 15, 2, 0, 0],
         [3, 15, 0, 0, 13, 0],
         [0, 2, 0, 0, 9, 10],
         [0, 0, 13, 9, 0, 12],
         [0, 0, 0, 10, 12, 0]]
start = 0
print(prim(graph, start))
```

Algoritmo de Kruskal

El algoritmo de Kruskal busca encontrar un árbol de expansión mínima (Minimum Spanning Tree, MST) para un grafo conectado, ponderado y no dirigido. El objetivo es conectar todos los nodos con la menor distancia total posible sin formar ciclos.

```
def find(parent, i):
    if parent[i] == i:
        return i
    return find(parent, parent[i])

def kruskal(graph):
    result = []  # Almacena el resultado del MST
    i, e = 0, 0  # Contadores de índices para los nodos y aristas del MST

    # Ordena todas las aristas en orden no decreciente por su peso
    graph = sorted(graph, key=lambda item: item[2])

    parent = [] ; rank = []

    for node in range(len(graph)):
        parent.append(node)
        rank.append(0)

    while e < len(graph) - 1:
        u, v, w = graph[i]
        i += 1
        x = find(parent, u)
        y = find(parent, v)

        # Si incluir esta arista no causa un ciclo, inclúyela en el
```

```
resultado
        # y aumenta el índice de la arista del MST
        if x != y:
            e += 1
            result.append((u, v, w))

    return result

# Ejemplo de un grafo
graph = [
    # (origen, destino, peso)
    (0, 1, 10),
    (0, 2, 6),
    (0, 3, 5),
    (1, 3, 15),
    (2, 3, 4)
]

# Ejecutamos el algoritmo
kruskal(graph)
```

Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford calcula los caminos más cortos desde un solo nodo fuente a todos los demás nodos en un grafo dirigido con pesos que pueden ser negativos.

```
def bellman_ford(graph, source):
    distance = {vertex: float('infinity') for vertex in graph}
    predecessor = {vertex: None for vertex in graph}

    distance[source] = 0

    for _ in range(len(graph) - 1):
        for u, v, w in graph:
            if distance[u] != float('infinity') and distance[u] + w <
distance[v]:
                distance[v] = distance[u] + w
                predecessor[v] = u

    # Verificar ciclos negativos
    for u, v, w in graph:
```

```
        if distance[u] != float('infinity') and distance[u] + w <
distance[v]:
            print("El grafo contiene un ciclo de peso negativo")
            return

    return distance, predecessor
```

Búsqueda en Anchura (BFS)

La búsqueda en anchura (Breadth-First Search, BFS) es un algoritmo para recorrer o buscar en estructuras de datos de grafo y se realiza nivel por nivel.

```
from collections import deque

def bfs(graph, start_vertex):
    visited = set()
    queue = deque([start_vertex])

    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            visited.add(vertex)

            # Añade vecinos no visitados a la cola
            queue.extend(set(graph[vertex]) - visited)

    return visited
```

Búsqueda en Profundidad (DFS)

La búsqueda en profundidad (Depth-First Search, DFS) es otro algoritmo de recorrido de grafos que sigue un camino hasta el final antes de retroceder a otros caminos.

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    for next_vertex in graph[start] - visited:
        dfs(graph, next_vertex, visited)
```

```
    return visited
```

Estos algoritmos son pilares fundamentales en la teoría de grafos y se aplican en una amplia variedad de campos, incluyendo la optimización de redes, la planificación de rutas y la organización de datos.

1.2.6 Archivos

Para tu examen sobre la organización de archivos en sistemas de bases de datos, es importante comprender tanto los conceptos teóricos como su implementación práctica. A continuación, te ofrezco una explicación detallada junto con un ejemplo de código para ilustrar estos conceptos.

Características y Tipos de Organización de Archivos

Características Principales

1. **Persistencia**: Los archivos son una manera duradera de almacenar datos, manteniéndolos accesibles incluso después de finalizar la ejecución del programa que los generó.
2. **Gran Volumen**: Capacidad para almacenar grandes cantidades de información.
3. **Acceso**: Puede ser secuencial (accediendo a los datos en el orden en que están almacenados) o directo (accediendo a los datos de manera no secuencial, generalmente más rápida).
4. **Estructura**: Los datos pueden estar estructurados (como en bases de datos) o no estructurados (como en documentos de texto o archivos multimedia).
5. **Metadata**: Información adicional sobre los archivos como su estructura, permisos, y ubicación.
6. **Seguridad**: Mecanismos para proteger los datos y controlar el acceso.

Tipos de Organización de Archivos

1. **Secuencial**: Los datos se almacenan uno detrás de otro. Es simple, pero la búsqueda puede ser lenta.
2. **Indexada**: Usa índices para una búsqueda más rápida.
3. **Hashing (Tablas Hash)**: Utiliza una función de hash para una localización y acceso rápido de los registros.

4. **Árboles B/B+**: Estructura de árbol balanceado para operaciones eficientes. En árboles B+, todas las llaves están en las hojas, lo que es ideal para recorridos secuenciales rápidos.

Operaciones en Archivos

- **Inserción**: Añadir nuevos registros, variando en complejidad según la organización del archivo.
- **Eliminación**: Remover registros, que puede implicar marcar registros como borrados o reorganizar el archivo.
- **Modificación**: Actualizar registros existentes.
- **Búsqueda**: Encontrar registros específicos, con un rendimiento que depende de la organización del archivo.

Ejemplo de Código: Hashing en Python

Vamos a ver un ejemplo de cómo implementar una simple tabla hash en Python. Este ejemplo ilustrará cómo se pueden realizar inserciones, búsquedas y eliminaciones en una tabla hash.

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def get_hash(self, key):
        return hash(key) % self.size

    def add(self, key, value):
        hash_key = self.get_hash(key)
        self.table[hash_key] = value

    def get(self, key):
        hash_key = self.get_hash(key)
        return self.table[hash_key]

    def remove(self, key):
        hash_key = self.get_hash(key)
        self.table[hash_key] = None

# Ejemplo de uso
```

```
ht = HashTable(10)
ht.add("clave1", "valor1")
ht.add("clave2", "valor2")
print(ht.get("clave1")) # Devuelve 'valor1'
ht.remove("clave1")
print(ht.get("clave1")) # Devuelve None
```

Este código define una clase `HashTable` que implementa una tabla hash básica con funciones de añadir, obtener y eliminar elementos. La función `get_hash` genera un índice hash para cada clave, lo que permite un acceso rápido a los elementos del arreglo.

Conclusión para el Estudio

Es fundamental entender cómo cada tipo de organización de archivos influye en la eficiencia de diferentes operaciones. Por ejemplo, mientras que las tablas hash son excelentes para accesos rápidos, los árboles B/B+ son más adecuados para mantener datos ordenados y realizar recorridos eficientes. Estos conceptos son clave para el diseño y optimización de sistemas de bases de datos.

1.2.7 Ordenamiento y Búsqueda

Los algoritmos de ordenamiento y búsqueda son esenciales en informática para manejar eficientemente estructuras de datos. Aquí encontrarás una explicación detallada de algunos algoritmos clave junto con ejemplos de código que puedes estudiar para tu examen.

Algoritmos de Búsqueda

1. Búsqueda Secuencial

Explicación:
Este método implica revisar cada elemento de un arreglo o lista uno por uno hasta encontrar el elemento deseado.

- **Mejor caso (O(1))**: El elemento se encuentra al principio de la lista.
- **Peor caso (O(n))**: El elemento está al final de la lista o no está presente.

Código en Python:

```python
def busqueda_secuencial(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

# Ejemplo de uso
lista = [3, 5, 2, 4, 7]
print(busqueda_secuencial(lista, 4))  # Devuelve el índice 3
```

2. Búsqueda Binaria

Explicación:
Requiere que la lista esté ordenada. Divide repetidamente a la mitad la porción de la lista que podría contener al elemento, hasta que lo encuentra o hasta que la sublista se vacía.

- **Mejor caso (O(1))**: El elemento está en el medio del arreglo.
- **Peor caso (O(log n))**: El elemento está en uno de los extremos o no está presente.

Código en Python:

```python
def busqueda_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

# Ejemplo de uso
lista_ordenada = [1, 3, 5, 7, 9]
print(busqueda_binaria(lista_ordenada, 7))  # Devuelve el índice 3
```

Algoritmos de Ordenamiento

1. Ordenamiento Burbuja

Explicación:
Compara pares adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que no hay más intercambios necesarios.

- **Mejor caso (O(n))**: La lista ya está ordenada.
- **Peor caso (O(n²))**: La lista está en orden inverso.

Código en Python:

```
def ordenamiento_burbuja(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista

# Ejemplo de uso
lista = [64, 34, 25, 12, 22, 11, 90]
print(ordenamiento_burbuja(lista))
```

2. Ordenamiento por Inserción

Explicación:
Construye la lista ordenada un elemento a la vez, insertando cada elemento nuevo en su posición correcta.

- **Mejor caso (O(n))**: La lista ya está ordenada.
- **Peor caso (O(n²))**: La lista está en orden inverso.

Código en Python:

```
def ordenamiento_insercion(lista):
    for i in range(1, len(lista)):
        clave = lista[i]
        j = i - 1
        while j >=0 and clave < lista[j]:
            lista[j + 1] = lista[j]
            j -= 1
```

```
            lista[j + 1] = clave
    return lista

# Ejemplo de uso
lista = [12, 11, 13, 5, 6]
print(ordenamiento_insercion(lista))
```

3. Ordenamiento por Selección

Explicación:
Busca el elemento más pequeño y lo intercambia con el primer elemento, luego busca el siguiente más pequeño y lo intercambia con el segundo elemento, y así sucesivamente.

- **Mejor y peor caso ($O(n^2)$)**: Independiente del orden inicial de los elementos.

Código en Python:

```
def ordenamiento_seleccion(lista

):
    for i in range(len(lista)):
        min_idx = i
        for j in range(i + 1, len(lista)):
            if lista[min_idx] > lista[j]:
                min_idx = j
        lista[i], lista[min_idx] = lista[min_idx], lista[i]
    return lista

# Ejemplo de uso
lista = [64, 25, 12, 22, 11]
print(ordenamiento_seleccion(lista))
```

Estos algoritmos son fundamentales en la informática y entender su funcionamiento te ayudará en tu examen y en tus futuros proyectos de programación. Recuerda que el mejor algoritmo a utilizar dependerá del contexto específico, como el tamaño de los datos y si están previamente ordenados o no.

1.3. Matemáticas discretas

Definición:

- Es el estudio de estructuras matemáticas que son fundamentalmente discretas en lugar de continuas.
- **Conceptos clave:** Lógica proposicional, conjuntos, relaciones, funciones, combinatoria, grafos y árboles.
- **Uso en CS:** Base para análisis y diseño de algoritmos y estructuras de datos.

1.3.1 Conteo, permutaciones y convinaciones

Para tu examen, es importante comprender los principios básicos del conteo en matemáticas, ya que son fundamentales para resolver problemas relacionados con la probabilidad, la estadística y las ciencias de la computación. A continuación, te explico cada principio junto con ejemplos de código en Python para que puedas estudiarlos mejor.

Principios de Conteo

1. Regla de la Suma (Principio de Adición)

Explicación:
Se usa cuando tienes dos opciones mutuamente excluyentes. Por ejemplo, si tienes `a` formas de hacer algo y `b` formas de hacer otra cosa (y no puedes hacer ambas al mismo tiempo), entonces hay `a + b` formas totales de realizar una acción.

Ejemplo de Código:
Supongamos que tienes 3 camisetas y 2 pantalones. Las formas de elegir una prenda (camiseta o pantalón) serían 3 + 2.

```
camisetas = 3
pantalones = 2
formas_de_elegir = camisetas + pantalones
print(formas_de_elegir)  # Devuelve 5
```

2. Regla del Producto (Principio de Multiplicación)

Explicación:
Se aplica cuando tienes dos acciones sucesivas. Si hay `a` maneras de hacer algo y `b` maneras de hacer otra cosa después, entonces hay `a * b` maneras de realizar ambas acciones en secuencia.

Ejemplo de Código:
Si puedes elegir entre 3 camisetas y 2 pantalones, las combinaciones posibles de atuendos serían 3 * 2.

```
camisetas = 3
pantalones = 2
combinaciones_atuendo = camisetas * pantalones
print(combinaciones_atuendo)  # Devuelve 6
```

Progresiones Geométricas y Aritméticas

1. Progresión Aritmética (PA)

Explicación:
Es una secuencia donde la diferencia entre términos consecutivos es constante. La fórmula general es `a + (n - 1) * d`, donde `a` es el primer término, `n` es el número del término y `d` es la diferencia común.

Ejemplo de Código:
Calculemos el 10º término de una PA donde el primer término es 2 y la diferencia común es 3.

```
a = 2  # Primer término
d = 3  # Diferencia común
n = 10  # Término que queremos encontrar

decimo_termino = a + (n - 1) * d
print(decimo_termino)  # Devuelve 29
```

2. Progresión Geométrica (PG)

Explicación:
Es una secuencia donde cada término se obtiene multiplicando el anterior por una constante. La fórmula general es `a * r^(n-1)`, donde `a` es el primer término, `r` es la razón y `n` es el número del término.

Ejemplo de Código:
Calculemos el 5º término de una PG donde el primer término es 3 y la razón es 2.

```
a = 3  # Primer término
r = 2  # Razón
n = 5  # Término que queremos encontrar

quinto_termino = a * r**(n - 1)
print(quinto_termino)  # Devuelve 48
```

Principio de las Casillas (Principio del Palomar)

Explicación:
Si distribuyes más objetos que casillas, al menos una casilla debe contener más de un objeto.

Ejemplo de Código:
Supongamos que tienes 10 palomas y 9 casillas.

```
palomas = 10
casillas = 9

minimo_palomas_por_casilla = (palomas + casillas - 1) // casillas
print(minimo_palomas_por_casilla)  # Devuelve 2
```

El código proporcionado calcula el mínimo número de palomas por casilla dado un número de palomas y un número de casillas. La fórmula usada es:

$$\text{minimo\_palomas\_por\_casilla} = \frac{\text{palomas} + \text{casillas} - 1}{\text{casillas}}$$

Para los valores dados (`palomas = 10` y `casillas = 9`), el cálculo sería:

$$\text{minimo\_palomas\_por\_casilla} = \frac{10 + 9 - 1}{9} = \frac{18}{9} = 2$$

Por lo tanto, el resultado que imprimiría el código sería `2`, lo que significa que al menos una casilla debe contener al menos 2 palomas si se distribuyen 10 palomas en 9 casillas.

Permutaciones y Combinaciones

Permutaciones

Explicación:
Las permutaciones se refieren al número de formas en que se pueden ordenar `r` elementos de un conjunto total de `n` elementos. La fórmula para calcular permutaciones es `n! / (n - r)!`, donde `n!` es el factorial de `n` y `(n - r)!` es el factorial de `n - r`.

Ejemplo de Código en Python:
Imagina que quieres saber de cuántas maneras diferentes puedes elegir y ordenar 3 libros de una estantería que tiene 5 libros en total.

```
import math

n = 5  # Total de libros
r = 3  # Libros a seleccionar y ordenar

permutaciones = math.factorial(n) / math.factorial(n - r)
print(permutaciones)  # Resultado
```

Combinaciones

Explicación:
Las combinaciones se refieren al número de formas en que se pueden seleccionar `r` elementos de un conjunto de `n` elementos, sin tener en cuenta el orden. La fórmula para calcular combinaciones es `n! / (r! * (n - r)!)`.

Ejemplo de Código en Python:
Si ahora quieres saber de cuántas maneras diferentes puedes elegir 3 libros de la misma estantería de 5 libros, pero el orden no importa, usarías la fórmula de combinaciones.

```
import math

n = 5  # Total de libros
r = 3  # Libros a seleccionar

combinaciones = math.factorial(n) / (math.factorial(r) * math.factorial(n - r))
print(combinaciones)  # Resultado
```

En estos ejemplos, `math.factorial` es una función de la biblioteca estándar de Python que calcula el factorial de un número. Las permutaciones dan un número mayor que las combinaciones para el mismo `n` y `r`, ya que las permutaciones tienen en cuenta el orden mientras que las combinaciones no.

1.3.2 Métodos de Demostración

Para tu examen, es crucial comprender los diferentes métodos de demostración matemática, ya que son herramientas fundamentales para validar teorías y resolver problemas complejos. A continuación, te proporciono una explicación de cada tipo de demostración, junto con ejemplos y, cuando sea posible, código en Python para ilustrar los conceptos.

1. Demostración Directa

Explicación:
Partes de hipótesis conocidas y usas lógica y teoremas previos para llegar directamente a una conclusión.

Ejemplo:
Para demostrar que "Si un número es par, entonces su cuadrado también es par", se parte del hecho de que un número par puede escribirse como `2n`, donde `n` es un número entero. Entonces, `(2n)^2 = 4n^2`, que es claramente un número par, ya que es divisible por 2.

Código en Python:

```
def es_par(n):
    return n % 2 == 0

def cuadrado_es_par(n):
    return es_par(n ** 2)

# Ejemplo de uso
assert cuadrado_es_par(4)  # No arroja error porque 4 al cuadrado (16) es
par
```

2. Demostración por Contrapositiva

Explicación:
Demuestras que si el consecuente es falso, entonces el antecedente también debe ser falso. Es útil cuando la contrapositiva es más fácil de probar que la declaración original.

Ejemplo:
Para probar que "Si un número no es divisible por 4, entonces no es divisible por 2", consideramos la contrapositiva: "Si un número es divisible por 2, entonces es divisible por 4". Esta afirmación es claramente falsa, por lo que la afirmación original también lo es.

3. Demostración por Contradicción

Explicación:
Supones que la afirmación que quieres probar es falsa y luego muestras que esta suposición lleva a una contradicción.

Ejemplo:
La demostración de que "√2 es irracional" es un clásico ejemplo de demostración por contradicción. Supones que √2 es racional (es decir, puede escribirse como una fracción de enteros a/b) y luego muestras que esto lleva a una contradicción.

4. Demostración por Inducción

Explicación:
Usada principalmente para afirmaciones sobre números enteros. Primero pruebas un caso base (como n = 1) y luego pruebas que si la afirmación es verdadera para n, también lo es para n + 1.

Ejemplo con Código en Python:
Para demostrar que "1 + 2 + 3 + ... + n = n(n + 1)/2", usamos inducción matemática.

```python
def suma_numeros(n):
    return n * (n + 1) // 2

# Caso base
assert suma_numeros(1) == 1

# Paso inductivo
n = 5
assert suma_numeros(n) == sum(range(1, n + 1))
```

5. Demostración por Ejemplo y Contraejemplo

Explicación:
Un ejemplo puede probar una afirmación existencial, mientras que un contraejemplo puede refutar una afirmación universal.

Ejemplo:
Un solo número primo par (2) es suficiente para demostrar que existen números primos pares. Sin embargo, cualquier número par mayor que 2 puede servir como contraejemplo para la afirmación "Todos los números pares son primos".

6. Demostración por Exhaustión

Explicación:
Divides el problema en casos y los pruebas uno por uno. Es útil cuando hay un número finito y manejable de casos.

Ejemplo:
Demostrar que un número entero es positivo, negativo o cero. Puedes probar cada caso por separado para llegar a una conclusión exhaustiva.

7. Demostración Constructiva vs. No Constructiva

Explicación:
Una demostración constructiva proporciona un método específico para encontrar un objeto matemático, mientras que una demostración no constructiva simplemente prueba la existencia sin proporcionar un método para encontrarlo.

Ejemplo:
Encontrar un número irracional entre dos racionales es un ejemplo de demostración constructiva, mientras que

la demostración de la existencia de una base de un espacio vectorial infinito dimensional es un ejemplo de una demostración no constructiva.

Estudiar estos tipos de demostraciones y entender cómo y cuándo aplicarlos te ayudará enormemente en tu examen y en tu carrera en matemáticas o ciencias de la computación.

1.3.3 Conjuntos

Los conjuntos son una de las nociones fundamentales en matemáticas y ciencias de la computación. Un conjunto es una colección de objetos distintos, considerados como un objeto en sí mismo. Los conjuntos se utilizan para agrupar y manejar objetos, permitiendo operaciones como la unión, intersección, diferencia, y verificación de pertenencia. A continuación, te proporciono una explicación detallada de los conceptos básicos de los conjuntos, junto con ejemplos de código en Python para que puedas estudiarlos para tu examen.

Conceptos Básicos de Conjuntos

1. **Definición de Conjunto:**
   Un conjunto es una colección de elementos sin un orden particular y sin elementos repetidos.
2. **Elementos de un Conjunto:**
   Cualquier cosa puede ser un elemento de un conjunto, como números, caracteres, o incluso otros conjuntos.
3. **Pertenencia:**
   Un elemento puede pertenecer o no a un conjunto, lo que se denota por los símbolos $\in$ (pertenece) y $\notin$ (no pertenece).
4. **Conjunto Vacío:**
   Un conjunto sin elementos se llama conjunto vacío, denotado por $\emptyset$.
5. **Subconjuntos:**
   Un conjunto A es un subconjunto de un conjunto B si todos los elementos de A están también en B, denotado por $A \subseteq B$.
6. **Operaciones de Conjuntos:**
   - **Unión (A ∪ B):** Conjunto de elementos que están en A, en B, o en ambos.
   - **Intersección (A ∩ B):** Conjunto de elementos que están tanto en A como en B.
   - **Diferencia (A - B):** Conjunto de elementos que están en A pero no en B.
   - **Complemento:** Conjunto de todos los elementos que no están en un conjunto dado.

Ejemplos de Código en Python

Vamos a usar Python para ilustrar algunas operaciones y conceptos básicos de conjuntos:

Creación de Conjuntos y Pertenencia

```python
# Crear conjuntos
A = {1, 2, 3}
B = {3, 4, 5}

# Comprobar si un elemento pertenece a un conjunto
print(2 in A)  # Devuelve True
print(2 in B)  # Devuelve False
```

Operaciones de Conjuntos

```python
# Unión de conjuntos
union = A.union(B)  # También se puede usar A | B
print(union)  # Devuelve {1, 2, 3, 4, 5}

# Intersección de conjuntos
interseccion = A.intersection(B)  # También se puede usar A & B
print(interseccion)  # Devuelve {3}

# Diferencia de conjuntos
diferencia = A.difference(B)  # También se puede usar A - B
print(diferencia)  # Devuelve {1, 2}
```

Subconjuntos

```python
# Crear subconjuntos
C = {1, 2}

# Comprobar si un conjunto es subconjunto de otro
print(C.issubset(A))  # Devuelve True
print(C.issubset(B))  # Devuelve False
```

Estudiar estos conceptos y ejemplos te ayudará a comprender cómo se utilizan los conjuntos en matemáticas y ciencias de la computación. Los conjuntos son particularmente importantes en el análisis de algoritmos y estructuras de datos, donde se utilizan para manejar y organizar colecciones de datos.

1.3.4 Funciones

El concepto de funciones es fundamental tanto en matemáticas como en ciencias de la computación. Una función describe una relación entre un conjunto de entradas y un conjunto de salidas posibles. En matemáticas, se define una función como una relación entre un conjunto denominado dominio y otro llamado codominio, de tal manera que a cada elemento del dominio le corresponde un único elemento del codominio. En programación, las funciones son bloques de código que realizan una tarea específica y pueden ser reutilizadas en diferentes partes de un programa.

Conceptos Básicos de Funciones

1. **Dominio y Codominio:**
   - **Dominio:** Conjunto de todos los valores de entrada posibles para la función.
   - **Codominio:** Conjunto de todos los posibles resultados o salidas de la función.
2. **Imagen o Rango:**
   Conjunto de todos los valores que la función efectivamente produce, siempre parte del codominio.
3. **Regla de Correspondencia:**
   La regla que define la relación entre los elementos del dominio y los del codominio.
4. **Tipos de Funciones en Matemáticas:**
   - **Inyectiva (Uno a Uno):** Si elementos diferentes del dominio se mapean a elementos diferentes en el codominio.
   - **Sobreyectiva (Sobre):** Si cada elemento del codominio es la imagen de al menos un elemento del dominio.
   - **Biyectiva (Uno a Uno y Sobre):** Si es tanto inyectiva como sobreyectiva.
5. **Funciones en Programación:**
   En programación, una función es una secuencia de declaraciones que se agrupan y pueden ser llamadas en diferentes partes del programa.

Ejemplos de Código en Python

Vamos a ilustrar el concepto de funciones en Python, tanto en el contexto matemático como en la programación:

Ejemplo Matemático: Función Cuadrática

```python
def funcion_cuadratica(x):
    return x**2

# Uso de la función
print(funcion_cuadratica(3))  # Devuelve 9
print(funcion_cuadratica(-4)) # Devuelve 16
```

Ejemplo de Función en Programación: Suma

```python
def suma(a, b):
    return a + b

# Llamada a la función
resultado = suma(5, 3)
print(resultado)  # Devuelve 8
```

Verificación de Tipo de Función (Inyectiva, Sobreyectiva, Biyectiva)

Para verificar si una función es inyectiva, sobreyectiva o biyectiva, normalmente se realiza un análisis matemático. En programación, esto puede ser más complejo y depende de la naturaleza específica de la función.

Conclusión para el Estudio

Entender las funciones es crucial para resolver problemas tanto en matemáticas como en programación. En matemáticas, te permite comprender cómo diferentes valores están relacionados entre sí, mientras que en programación, te ayuda a estructurar el código de manera más eficiente y reutilizable. Estudiar ejemplos de funciones y practicar con ejercicios te ayudará a profundizar tu comprensión de este importante concepto.

1.3.5 Operaciones Aritméticas en Diferentes Bases Numéricas

Las operaciones aritméticas en diferentes bases numéricas son fundamentales en la computación, especialmente porque las computadoras operan utilizando el sistema binario (base 2). Además, los sistemas octal (base 8) y hexadecimal (base 16) son también comúnmente utilizados en informática debido a que proporcionan una manera más compacta y legible de representar datos binarios. A continuación,

describo cómo se realizan las operaciones aritméticas básicas en estas bases numéricas:

Sistema Binario (Base 2)

Suma

Para sumar dos números en sistema binario, se suman los dígitos de igual peso (como en base 10), pero se tiene en cuenta que (1 + 1 = 10) en binario, por lo que el '1' se lleva a la siguiente columna.

Ejemplo:

```
   1011
 + 1101
-------
  11000
```

Resta

Para restar en binario, se utiliza el préstamo, similar a la resta en base 10, pero cuando se toma prestado se suma 2 al dígito actual en vez de 10.

Ejemplo:

```
   1011
 -  100
-------
    111
```

Multiplicación

La multiplicación se realiza de manera similar a la base 10, multiplicando dígito a dígito y sumando los resultados parciales, teniendo en cuenta que en binario (1 \times 1 = 1) y cualquier dígito multiplicado por 0 da 0.

Ejemplo:

```
    101
  x  11
-------
    101
   101
-------
   1111
```

Sistema Octal (Base 8)

Las operaciones en base 8 son similares a las de base 10, pero cada dígito puede ser un valor de 0 a 7.

Suma

Se suman los dígitos de igual peso y se lleva una unidad cada vez que la suma de dos dígitos supera 7.

Ejemplo:

```
  157
+  64
-----
  243
```

Resta

Se realiza igual que en base 10, con la diferencia de que al pedir prestado se suma 8 al dígito.

Ejemplo:

```
  720
-  35
-----
  663
```

Sistema Hexadecimal (Base 16)

En hexadecimal, cada dígito cuenta 16 veces el valor del dígito a su derecha. Los números van del 0 al 9 y luego se usan las letras A a F para representar los valores 10 a 15.

Suma

La suma se realiza igual que en las otras bases, con el cuidado de convertir de hexadecimal a decimal para sumar y luego volver a hexadecimal si es necesario.

Ejemplo:

```
   1A3
 +  BF
-----
  262
```

Resta

Al igual que en la suma, puede ser necesario convertir los valores para restar y pedir prestado, añadiendo 16 en lugar de 10.

Ejemplo:

```
  2A0
-  B9
-----
  1E7
```

Para realizar estas operaciones con éxito, es esencial familiarizarse con las tablas de suma y multiplicación de estas bases, así como con la conversión entre bases numéricas. En la programación y la electrónica, estas operaciones son frecuentemente realizadas por hardware y software especializado, y son una parte fundamental de los sistemas de cómputo modernos.

1.3.6 Operaciones con Matrices

Las operaciones con matrices son fundamentales en muchas áreas de estudio y tienen aplicaciones prácticas en campos como la física, la ingeniería y la informática. A continuación, te explico las operaciones básicas con matrices y proporciono

ejemplos de código en Python para cada una de ellas, lo cual te será útil para tu examen.

Suma y Resta de Matrices

La suma y la resta de matrices se realizan elemento por elemento, y solo es posible si ambas matrices tienen las mismas dimensiones.

Suma de Matrices

Si `A` y `B` son matrices de dimensiones `m x n`, entonces la suma `(A + B)ij = Aij + Bij`.

Resta de Matrices

Para la resta, la operación es similar: `(A - B)ij = Aij - Bij`.

Ejemplo de Código en Python:

```
import numpy as np

# Suma de matrices
def suma_matrices(A, B):
    return np.add(A, B)

# Resta de matrices
def resta_matrices(A, B):
    return np.subtract(A, B)

# Ejemplo de uso
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print("Suma:\n", suma_matrices(A, B))
print("Resta:\n", resta_matrices(A, B))
```

Multiplicación de Matrices

La multiplicación de matrices requiere que el número de columnas de la primera matriz sea igual al número de filas de la segunda.

Ejemplo de Código en Python:

```python
# Multiplicación de matrices
def multiplicacion_matrices(A, B):
    return np.dot(A, B)

# Ejemplo de uso
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print("Multiplicación:\n", multiplicacion_matrices(A, B))
```

Multiplicación por un Escalar

Multiplicar cada elemento de la matriz `A` por un escalar `k`.

Ejemplo de Código en Python:

```python
# Multiplicación por un escalar
def escalar_por_matriz(k, A):
    return k * A

# Ejemplo de uso
k = 3
A = np.array([[1, 2], [3, 4]])

print("Escalar por Matriz:\n", escalar_por_matriz(k, A))
```

Determinante de una Matriz

El determinante es una propiedad de las matrices cuadradas y se denota como `det(A)` o `|A|`.

Ejemplo de Código en Python:

```python
# Determinante de una matriz
def determinante(A):
    return np.linalg.det(A)

# Ejemplo de uso
A = np.array([[1, 2], [3, 4]])

print("Determinante:", determinante(A))
```

Matriz Inversa

Una matriz `A` es invertible si existe `B` tal que `AB = BA = I`, donde `I` es la matriz identidad.

Ejemplo de Código en Python:

```python
# Matriz inversa
def matriz_inversa(A):
    return np.linalg.inv(A)

# Ejemplo de uso
A = np.array([[1, 2], [3, 4]])

print("Inversa:\n", matriz_inversa(A))
```

Eigenvalores y Eigenvectores

Un vector `v` es un eigenvector de una matriz `A` si `A * v = λ * v`, donde `λ` es un eigenvalor correspondiente a `v`.

Ejemplo de Código en Python:

```python
# Eigenvalores y eigenvectores
def eigen(A):
    return np.linalg.eig(A)

# Ejemplo de uso
A = np.array([[1, 2], [3, 4]])

eigenvalores, eigenvectores = eigen(A)
print("Eigenvalores:\n", eigenvalores)
print("Eigenvectores:\n", eigenvectores)
```

Estos ejemplos te proporcionarán una base sólida para entender las operaciones con matrices. Recuerda que la biblioteca `numpy` en Python es muy útil para realizar operaciones matriciales de manera eficiente. Estudiar estas operaciones y practicar con ejemplos te ayudará a prepararte para tu examen y para aplicaciones prácticas en tu carrera.

1.4. Lógica computacional

La lógica computacional es un área de las ciencias de la computación y matemáticas que utiliza la lógica para tratar problemas relacionados con la computación. Aquí hay algunos aspectos clave y conceptos básicos asociados con la lógica computacional:

1. Lógica Proposicional:

Es el estudio de las proposiciones y su composición a través de operadores lógicos como AND (conjunción), OR (disyunción), NOT (negación), IF...THEN (implicación) y IF AND ONLY IF (bicondicional).

2. Lógica de Primer Orden:

También conocida como lógica de predicados, extiende la lógica proposicional al incluir cuantificadores como "para todo" (universal) y "existe" (existencial), así como relaciones y funciones.

3. Sistemas Formales:

Un sistema formal consiste en un lenguaje formal y un sistema de deducción que puede incluir axiomas y reglas de inferencia para construir demostraciones.

4. Inferencia:

Se refiere al proceso de llegar a conclusiones lógicas a partir de premisas conocidas. Las reglas de inferencia definen cómo se pueden formar nuevas proposiciones a partir de otras ya existentes.

5. Semántica:

Mientras la sintaxis se refiere a la forma de las expresiones lógicas, la semántica trata su significado. En lógica computacional, se analiza el significado de las expresiones y su correspondencia con estados de la realidad o modelos matemáticos.

6. Satisfactibilidad:

Una fórmula es satisfacible si existe al menos una asignación de valores de verdad a sus variables que la hace verdadera. La satisfactibilidad es un concepto central en la teoría de la computación, particularmente en problemas de decisión y en la verificación de modelos.

7. Teorema de Completitud de Gödel:

Establece que si un sistema de lógica de primer orden es consistente (no contiene contradicciones), entonces todas las verdades que se pueden formular en el sistema pueden ser demostradas dentro del mismo.

8. Teorema de Incompletitud de Gödel:

Este teorema muestra que para cualquier sistema formal suficientemente rico en aritmética, existen proposiciones verdaderas que no pueden ser probadas dentro del sistema.

9. Lógica Temporal:

Es una extensión de la lógica que se utiliza para razonar sobre el orden temporal de los eventos y se aplica comúnmente en la verificación de software y sistemas.

10. Lógica Modal:

Incluye operadores modales que califican proposiciones respecto a la posibilidad o necesidad. En la informática, se usa para razonar sobre conocimiento, creencias y obligaciones.

11. Resolución:

Es una regla de inferencia utilizada en lógica de primer orden que permite derivar conclusiones a partir de cláusulas contradictorias. Es la base de muchos algoritmos de prueba automática de teoremas y sistemas de verificación de modelos.

La lógica computacional es la base de la programación, la verificación de software, la inteligencia artificial y la base de datos, entre otras áreas. Comprender estos conceptos es fundamental para avanzar en el estudio de la teoría de la computación y los sistemas formales.

1.4.1 Lógica Proposicional

La lógica proposicional es un pilar fundamental en matemáticas, informática y filosofía, y proporciona un marco para razonar sobre declaraciones verdaderas o falsas. A continuación, exploramos los conceptos clave de la lógica proposicional y

proporcionamos ejemplos de código en Python para ilustrar estos conceptos, lo que te será de gran ayuda para tu examen.

Elementos Fundamentales

- **Proposiciones Atómicas:** Son declaraciones básicas que pueden ser verdaderas o falsas. Ejemplos: "Está lloviendo", "El sol es una estrella".
- **Operadores Lógicos:** Permiten formar proposiciones más complejas a partir de proposiciones atómicas.
  - **Conjunción (AND, ∧):** Verdadera si ambas proposiciones son verdaderas.
  - **Disyunción (OR, ∨):** Verdadera si al menos una de las proposiciones es verdadera.
  - **Negación (NOT, ¬):** Invierte el valor de verdad de una proposición.
  - **Implicación (→):** Falsa solo si la primera proposición es verdadera y la segunda falsa.
  - **Equivalencia (↔):** Verdadera si ambas proposiciones tienen el mismo valor de verdad.

Tablas de Verdad en Python

Las tablas de verdad son herramientas que muestran el resultado de aplicar operadores lógicos a proposiciones.

Ejemplo de Código:

```
def AND(a, b):
    return a and b

def OR(a, b):
    return a or b

def NOT(a):
    return not a

def IMPLICA(a, b):
    return NOT(a) or b

def EQUIVALE(a, b):
    return IMPLICA(a, b) and IMPLICA(b, a)

# Uso de las funciones
```

```
print(AND(True, False))  # False
print(OR(True, False))   # True
print(NOT(True))         # False
print(IMPLICA(True, False)) # False
print(EQUIVALE(True, True)) # True
```

Formas Normales

- **Forma Normal Conectiva (FNC):** Una proposición en FNC se forma con conjunciones (ANDs) de disyunciones (ORs) de literales.
- **Forma Normal Disyuntiva (FND):** Una proposición en FND se forma con disyunciones (ORs) de conjunciones (ANDs) de literales.

Validez de una Fórmula Bien Formada

- **Tautología:** Siempre verdadera, independientemente de los valores de verdad de sus componentes.
- **Contradicción:** Siempre falsa.
- **Contingencia:** Puede ser verdadera o falsa dependiendo de los valores de verdad de sus componentes.

Reglas de Inferencia Proposicional

Estas reglas son utilizadas para derivar nuevas proposiciones a partir de otras ya conocidas.

Lógica de Predicados y Cuantificadores

Amplía la lógica proposicional para hablar sobre propiedades de objetos y relaciones entre ellos.

Álgebra de Boole

Se utiliza para manipular expresiones lógicas y es fundamental en el diseño de circuitos digitales y sistemas de computación.

Este resumen te proporciona una base sólida para entender la lógica proposicional. Estudiar estos conceptos y practicar con ejercicios de codificación te ayudará a prepararte para tu examen y a desarrollar habilidades críticas en el razonamiento lógico.

1.4.2 Conectores Lógicos y Tablas de Verdad

Los conectores lógicos son símbolos que se utilizan para conectar proposiciones en la lógica proposicional y formar proposiciones más complejas. Aquí hay una breve descripción de los principales conectores y sus tablas de verdad asociadas:

Conectores Lógicos Principales

1. **Conjunción (AND, ∧)**:
   - Sólo es verdadera si ambas proposiciones son verdaderas.
   - Ejemplo: "p ∧ q" es verdadera si tanto "p" como "q" son verdaderas.
2. **Disyunción (OR, ∨)**:
   - Es verdadera si al menos una de las proposiciones es verdadera.
   - Ejemplo: "p ∨ q" es verdadera si "p" o "q" (o ambas) son verdaderas.
3. **Negación (NOT, ¬)**:
   - Invierte el valor de verdad de una proposición.
   - Ejemplo: "¬p" es verdadera si "p" es falsa.
4. **Implicación (→, =>)**:
   - Indica que si la primera proposición es verdadera, entonces la segunda también debe serlo para que toda la expresión sea verdadera.
   - Ejemplo: "p → q" es falsa sólo si "p" es verdadera y "q" es falsa.
5. **Bicondicional (↔, <=>)**:
   - Indica que ambas proposiciones tienen el mismo valor de verdad.
   - Ejemplo: "p ↔ q" es verdadera si "p" y "q" son ambas verdaderas o ambas falsas.

Tablas de Verdad

La tabla de verdad para cada conector muestra todas las combinaciones posibles de los valores de verdad para las proposiciones y el resultado de aplicar el conector.

p	**q**	**p ∧ q**	**p ∨ q**	**¬p**	**p → q**	**p ↔ q**
V (Verdadero)	V	V	V	F	V	V
V	F	F	V	F	F	F
F	V	F	V	V	V	F
F	F	F	F	V	V	V

- **V**: Verdadero
- **F**: Falso

Ejercicio 1: Combinando AND y OR

Dadas dos proposiciones `p` y `q`, escribe la tabla de verdad para la expresión `(p AND q) OR (NOT p)`.

Ejercicio 2: Implicación y Negación

Considera las proposiciones `p` y `q`. Escribe la tabla de verdad para `NOT (p → q)`.

Ejercicio 3: Bicondicional

Para las proposiciones `p` y `q`, construye la tabla de verdad para `p ↔ q` y compara los resultados con las tablas de verdad para `p → q` y `q → p`.

Ejercicio 4: Combinaciones Complejas

Crea la tabla de verdad para la expresión `(p OR NOT q) AND (q OR NOT p)`.

Ejercicio 5: Desafío con Tautología

Demuestra que `p OR NOT p` es una tautología.

Ejercicio 6: Explorando Contradicciones

Muestra que `p AND NOT p` es una contradicción.

Ejercicio 7: Leyes de De Morgan

Usa las leyes de De Morgan para escribir las tablas de verdad de `NOT (p AND q)` y `NOT (p OR q)` y compáralas con las tablas de verdad de `(NOT p) OR (NOT q)` y `(NOT p) AND (NOT q)` respectivamente.

Ejercicio 8: Implicación en Términos de OR y NOT

Demuestra que `p → q` es lógicamente equivalente a `NOT p OR q`.

Ejercicio 9: Equivalencia Compleja

Evalúa si la expresión `(p → q) AND (q → p)` es equivalente a `p ↔ q`.

Ejercicio 10: Combinando Todo

Evalúa la expresión `(p AND q) OR (NOT p AND NOT q)`. ¿Es una tautología, una contradicción, o ninguna de las dos?

Entiendo que deseas los resultados de las tablas de verdad para cada uno de los ejercicios mencionados. A continuación, te proporciono los resultados, pero te recomiendo intentar resolverlos primero por tu cuenta para reforzar tu aprendizaje.

Resultados de los Ejercicios

Ejercicio 1: Combinando AND y OR

`(p AND q) OR (NOT p)`

p	q	p AND q	NOT p	(p AND q) OR (NOT p)
V	V	V	F	V
V	F	F	F	F
F	V	F	V	V
F	F	F	V	V

Ejercicio 2: Implicación y Negación

`NOT (p → q)`

p	q	p → q	NOT (p → q)
V	V	V	F
V	F	F	V
F	V	V	F
F	F	V	F

Ejercicio 3: Bicondicional

`p ↔ q`

p	q	p ↔ q
V	V	V
V	F	F

p	q	p ↔ q
F	V	F
F	F	V

Ejercicio 4: Combinaciones Complejas

`(p OR NOT q) AND (q OR NOT p)`

p	q	NOT q	p OR NOT q	NOT p	q OR NOT p	(p OR NOT q) AND (q OR NOT p)
V	V	F	V	F	V	V
V	F	V	V	F	F	F
F	V	F	F	V	V	F
F	F	V	V	V	V	V

Ejercicio 5: Desafío con Tautología

`p OR NOT p` (Siempre verdadero)

p	NOT p	p OR NOT p
V	F	V
F	V	V

Ejercicio 6: Explorando Contradicciones

`p AND NOT p` (Siempre falso)

p	NOT p	p AND NOT p
V	F	F
F	V	F

Ejercicio 7: Leyes de De Morgan

`NOT (p AND q)` vs. `(NOT p) OR (NOT q)`

p	q	p AND q	NOT (p AND q)	NOT p	NOT q	(NOT p) OR (NOT q)
V	V	V	F	F	F	F
V	F	F	V	F	V	V

p	q	p AND q	NOT (p AND q)	NOT p	NOT q	(NOT p) OR (NOT q)
F	V	F	V	V	F	V
F	F	F	V	V	V	V

`NOT (p OR q)` vs. `(NOT p) AND (NOT q)`

p	q	p OR q	NOT (p OR q)	NOT p	NOT q	(NOT p) AND (NOT q)
V	V	V	F	F	F	F
V	F	V	F	F	V	F
F	V	V	F	V	F	F
F	F	F	V	V	V	V

Ejercicio 8: Implicación en Términos de OR y NOT

`p → q` vs. `NOT p OR q`

p	q	p → q	NOT p	NOT p OR q
V	V	V	F	V
V	F	F	F	F
F	V	V	V	V
F	F	V	V	V

Ejercicio 9: Equivalencia Compleja

`(p → q) AND (q → p)` vs. `p ⇔ q`

p	q	p → q	q → p	(p → q) AND (q → p)	p ↔ q
V	V	V	V	V	V
V	F	F	V	F	F
F	V	V	F	F	F
F	F	V	V	V	V

Ejercicio 10: Combinando Todo

`(p AND q) OR (NOT p AND NOT q)`

p	q	p AND q	NOT p	NOT q	NOT p AND NOT q	(p AND q) OR (NOT p AND NOT q)
V	V	V	F	F	F	V
V	F	F	F	V	F	F
F	V	F	V	F	F	F
F	F	F	V	V	V	V

En estos ejercicios, puedes ver cómo se aplican los conectores lógicos y cómo construir sus tablas de verdad. La práctica con estos ejercicios te ayudará a entender mejor la lógica proposicional, un elemento crucial en el razonamiento lógico y matemático.

1.4.3 Formas Normales Conectiva y Disyuntiva

Las Formas Normales Conectiva (FNC) y Disyuntiva (FND) son fundamentales en la lógica proposicional y se utilizan para simplificar y analizar proposiciones lógicas. Estas formas son cruciales para diversas aplicaciones en computación, como la verificación de satisfacibilidad, demostración automática de teoremas y simplificación de circuitos. Aquí te explico cómo entender y trabajar con estas formas normales, incluyendo ejemplos de código en Python para tu examen.

Forma Normal Conectiva (FNC)

Una fórmula está en FNC si se compone de una conjunción (AND) de cláusulas, donde cada cláusula es una disyunción (OR) de literales (una proposición atómica o su negación).

Ejemplo FNC:
La expresión `(p ∨ ¬q) ∧ (¬p ∨ r)` es un ejemplo de FNC.

Cómo convertir una expresión lógica a FNC:

1. Elimina las implicaciones: Convierte todas las implicaciones (`p → q`) en disyunciones equivalentes (`¬p ∨ q`).
2. Mueve las negaciones hacia adentro usando las leyes de De Morgan.
3. Distribuye la conjunción sobre la disyunción.

Ejemplo de Código en Python:

```
# Supongamos que tenemos literales p, q, r
p, q, r = True, False, True

# Expresión en FNC: (p ∨ ¬q) ∧ (¬p ∨ r)
FNC_expresion = (p or not q) and (not p or r)
print(FNC_expresion)  # Evalúa la expresión
```

Forma Normal Disyuntiva (FND)

Una fórmula está en FND si se compone de una disyunción (OR) de conjuntos, donde cada conjunto es una conjunción (AND) de literales.

Ejemplo FND:
La expresión `(p ∧ q) ∨ (¬r)` es un ejemplo de FND.

Cómo convertir una expresión lógica a FND:

1. Elimina las implicaciones como en FNC.
2. Mueve las negaciones hacia adentro.
3. Distribuye la disyunción sobre la conjunción.

Ejemplo de Código en Python:

```
# Supongamos que tenemos literales p, q, r
p, q, r = True, False, True

# Expresión en FND: (p ∧ q) ∨ (¬r)
FND_expresion = (p and q) or (not r)
print(FND_expresion)  # Evalúa la expresión
```

Utilidad de las Formas Normales

- **FNC:** Utilizada para comprobación de satisfacibilidad y procedimientos de resolución.
- **FND:** Usada en algoritmos de simplificación de circuitos y verificación de equivalencias lógicas.

Estudiar y practicar con estas formas normales te ayudará a comprender cómo se pueden simplificar y analizar proposiciones lógicas en la lógica computacional y te

preparará para tu examen. Además, estas habilidades son valiosas para aplicaciones en inteligencia artificial y diseño de circuitos.

1.4.4 Validez de una Fórmula Bien Formada (tautologías, contradicción)

Las formas normales son maneras de estructurar proposiciones lógicas en la lógica proposicional para facilitar ciertas operaciones, como la verificación de la satisfacibilidad o la comparación de equivalencias lógicas. Dos de las formas normales más importantes son la Forma Normal Conectiva (FNC) y la Forma Normal Disyuntiva (FND).

Forma Normal Conectiva (FNC)

Una fórmula lógica está en Forma Normal Conectiva si es una conjunción de cláusulas, donde cada cláusula es una disyunción de literales. Un literal es una proposición atómica o su negación. En términos más simples, la FNC es una conjunción (AND) de disyunciones (ORs).

La FNC se puede expresar como:

```
(C1 ∧ C2 ∧ ... ∧ Cn)
```

donde cada `Ci` es una cláusula de la forma:

```
(l1 ∨ l2 ∨ ... ∨ lm)
```

y cada `li` es un literal.

Por ejemplo, la expresión `(p ∨ ¬q) ∧ (¬p ∨ r)` está en FNC.

Forma Normal Disyuntiva (FND)

Una fórmula está en Forma Normal Disyuntiva si es una disyunción de conjuntos, donde cada conjunto es una conjunción de literales. Esencialmente, la FND es una disyunción (OR) de conjunciones (ANDs).

La FND se puede expresar como:

```
(D1 ∨ D2 ∨ ... ∨ Dn)
```

donde cada `Di` es un conjunto de la forma:

```
(l1 ∧ l2 ∧ ... ∧ lm)
```

y cada `li` es un literal.

Por ejemplo, la expresión `(p ∧ q) ∨ (¬r)` está en FND.

Utilidad de las Formas Normales

- **FNC**: Es útil para la comprobación de la satisfacibilidad y en los procedimientos de resolución, que son técnicas para la demostración automática de teoremas.
- **FND**: Se utiliza a menudo en algoritmos de simplificación de circuitos y también para verificar la equivalencia de fórmulas lógicas.

Ambas formas normales son herramientas clave en el campo de la lógica computacional y tienen aplicaciones prácticas en áreas como la verificación formal, la síntesis de circuitos y la inteligencia artificial.

1.4.5 Reglas de Inferencia Proposicional

Las reglas de inferencia proposicional son mecanismos lógicos que permiten derivar una conclusión a partir de premisas dadas. Son fundamentales para construir argumentos válidos en lógica y se utilizan ampliamente en la demostración de teoremas, tanto en matemáticas como en informática. A continuación, se detallan algunas de las reglas de inferencia más comunes:

Modus Ponens (MP)

Si "p" implica "q" (p → q) y "p" es verdadero, entonces "q" también debe ser verdadero.

```
p → q
p
-----
∴ q
```

Modus Tollens (MT)

Si "p" implica "q" (p → q) y "q" es falso, entonces "p" también debe ser falso.

```
p → q
¬q
-----
∴ ¬p
```

Silogismo Hipotético (SH)

Si "p" implica "q" (p → q) y "q" implica "r" (q → r), entonces "p" implica "r" (p → r).

```
p → q
q → r
-----
∴ p → r
```

Silogismo Disyuntivo (SD)

Si tenemos una disyunción "p o q" (p ∨ q) y sabemos que "p" es falso, entonces "q" debe ser verdadero.

```
p ∨ q
¬p
-----
∴ q
```

Conjunción (CON)

Si "p" es verdadero y "q" es verdadero, entonces "p y q" (p ∧ q) es verdadero.

```
p
q
-----
∴ p ∧ q
```

Simplificación (SIMP)

Si "p y q" (p ∧ q) es verdadero, entonces "p" es verdadero.

```
p ∧ q
-----
∴ p
```

Adición (ADD)

Si "p" es verdadero, entonces "p o q" (p ∨ q) es verdadero.

```
p
-----
∴ p ∨ q
```

Doble Negación (DN)

Si "p" es verdadero, entonces "no no p" (¬¬p) también es verdadero, y viceversa.

```
¬¬p
-----
∴ p
```

Estas reglas son aplicadas en un marco de trabajo lógico donde las proposiciones se manejan de manera sintáctica. En la lógica proposicional, estas reglas garantizan que si las premisas son verdaderas, las conclusiones derivadas también lo serán, preservando la veracidad a través de los argumentos.

1.4.5 Lógica de Predicados y Cuantificadores

La lógica de predicados, también conocida como lógica de primer orden, amplía la lógica proposicional al incluir cuantificadores y predicados que permiten la expresión de afirmaciones más complejas sobre algún dominio de discurso. A diferencia de la lógica proposicional, que trata con proposiciones simples, la lógica de predicados puede discutir relaciones entre objetos y la existencia o no de ciertos objetos.

Predicados

Un predicado es una función que toma un conjunto de objetos y devuelve un valor de verdad. Por ejemplo, en la lógica de predicados, podemos expresar propiedades o relaciones como P(x), donde P es un predicado y x es un objeto en el dominio de discurso.

Cuantificadores

Los cuantificadores son operadores que permiten hacer afirmaciones sobre los términos, como la existencia o la universalidad de ellos dentro del dominio de discurso. Hay dos cuantificadores principales en la lógica de predicados:

1. **Cuantificador Universal (∀)** - Afirma que la propiedad o relación es verdadera para todos los elementos del dominio de discurso. Por ejemplo, "∀x P(x)" se lee como "para todo x, P(x) es verdadero", o más simplemente, "todos los x son P".
2. **Cuantificador Existencial (∃)** - Afirma que hay al menos un elemento en el dominio de discurso para el cual la propiedad o relación es verdadera. Por ejemplo, "∃x P(x)" se lee como "existe al menos un x tal que P(x) es verdadero".

Ejemplo de Uso de Predicados y Cuantificadores

Supongamos que P(x) representa "x es un programador". Podemos usar la lógica de predicados para expresar las siguientes afirmaciones:

- ∀x P(x): "Todos son programadores."
- ∃x P(x): "Hay al menos un programador."
- ∀x ¬P(x): "Nadie es programador."
- ∃x ¬P(x): "Hay al menos una persona que no es programadora."

Reglas de Inferencia en la Lógica de Predicados

Las reglas de inferencia que se aplican en la lógica de predicados incluyen, pero no se limitan a:

- **Generalización Universal**: Si una afirmación es verdadera para un caso arbitrario, entonces es verdadera para todos los casos.

```
P(a)
-----
∴ ∀x P(x)
```

- **Instanciación Existencial**: Si algo es verdadero universalmente, entonces es verdadero para algún elemento particular.

```
∃x P(x)
-----
∴ P(a)
```

- **Instanciación Universal**: Si se afirma que una propiedad es verdadera para todos los elementos, entonces esa propiedad es verdadera para cualquier elemento específico.

```
∀x P(x)
-----
∴ P(a)
```

- **Generalización Existencial**: Si una propiedad es verdadera para algún caso específico, entonces es verdadera para al menos un caso.

```
P(a)
-----
∴ ∃x P(x)
```

Importancia de la Lógica de Predicados

La lógica de predicados es una herramienta poderosa en matemáticas y ciencias de la computación, ya que permite la formalización de teorías y la verificación de la veracidad de las afirmaciones dentro de esas teorías. Es esencial en la demostración automática de teoremas, la programación lógica, las bases de datos y la inteligencia artificial.

1.4.6 Álgebra de Boole

El Álgebra de Boole, desarrollada por George Boole, es una piedra angular de la lógica moderna y la computación. Su aplicación se extiende desde el diseño de circuitos digitales hasta la teoría de conjuntos y la informática. Aquí exploramos las

operaciones básicas, leyes y propiedades del Álgebra de Boole, así como su relevancia en la tecnología actual.

Operaciones Básicas

1. **AND (∧):** Es verdadero solo si ambos operandos son verdaderos. Esencial en la lógica de compuertas y en la evaluación de condiciones múltiples en programación.
2. **OR (∨):** Verdadero si al menos uno de los operandos es verdadero. Se utiliza en la toma de decisiones lógicas y en el diseño de circuitos.
3. **NOT (¬):** Invierte el valor de verdad. Fundamental en la lógica de programación y en el diseño de circuitos para alterar señales.

Leyes y Propiedades

- **Ley de Identidad:** Establece la conservación del valor de verdad.
- **Ley del Dominio:** Define los valores extremos de las operaciones.
- **Ley de Idempotencia:** Indica que repetir la operación no cambia el resultado.
- **Ley de Inversos:** Describe cómo se anulan mutuamente dos valores opuestos.
- **Ley de Complementos:** Se basa en la doble negación.
- **Ley Conmutativa:** Permite cambiar el orden de los operandos.
- **Ley Asociativa:** Facilita la agrupación de operaciones.
- **Ley Distributiva:** Combina operaciones de diferente naturaleza.
- **Leyes de De Morgan:** Permiten transformar conjunciones en disyunciones y viceversa.
- **Ley de Absorción:** Simplifica las expresiones combinadas.
- **Ley de Simplificación:** Reduce la complejidad de las expresiones.

Ejemplos de Código

Podemos ilustrar el Álgebra de Boole con ejemplos en Python, usando operadores lógicos para representar las operaciones de Boole:

```
# Definiendo variables booleanas
x = True
y = False

# AND
print("x AND y:", x and y)
```

```
# OR
print("x OR y:", x or y)

# NOT
print("NOT x:", not x)

# Leyes de De Morgan
print("NOT (x AND y):", not (x and y))
print("NOT x OR NOT y:", not x or not y)
```

Aplicaciones del Álgebra de Boole

El Álgebra de Boole es fundamental en:

- **Diseño de Circuitos Digitales:** Las puertas lógicas como AND, OR, NOT, etc., son la base de los circuitos digitales.
- **Programación de Computadoras:** Se utiliza para el control de flujo y la toma de decisiones.
- **Bases de Datos:** En la búsqueda y filtrado de datos.
- **Arquitectura de Computadoras:** En el diseño de CPU y sistemas de memoria.

Conclusión

El Álgebra de Boole es esencial para cualquiera en el campo de la computación o la ingeniería electrónica. Comprender estas leyes y operaciones no solo es crucial para la teoría, sino también para aplicaciones prácticas en diseño de hardware, programación y análisis de sistemas lógicos. Su estudio proporciona una base sólida para el razonamiento lógico y la resolución de problemas en tecnología e informática.